Help - Help for Webmasters

« back to results for ""**Excel by Example**""

Below is a cache of **http://www.chfi.unipd.it/unix/vb5/Pgch10.html**. It's a snapshot of the page taken as our search engine partner crawled the web. We've highlighted the words: **excel by example**
The website itself may have changed. You can check the current page (without highlighting). **Beta**

Yahoo! is not affiliated with the authors of this page or responsible for its content.

Chapter 10.

Programming with Components

Do you sometimes need to provide the same analysis and calculation capabilities as Microsoft Excel in your Visual Basic application? Or, perhaps you'd like to format a document using Microsoft Word formatting tools, or store and manage data using the Microsoft Jet database engine. Even better, would you like to be able to create or buy standard components, then use them in multiple applications without having to modify them?

All this and more can be accomplished by building your applications using ActiveX components. An *ActiveX component* is a reusable piece of programming code and data made up of one or more objects created using ActiveX technology. Your applications can use existing components, such as those included in Microsoft Office applications, code components, ActiveX documents, or ActiveX controls (formerly called OLE controls) provided by a variety of vendors. Or, if you have the Visual Basic, Professional or Enterprise Edition, you can create your own ActiveX controls.

For components that support object linking and embedding, you can insert objects into your application without writing any code by using the component's visual interface. You can insert an OLE-enabled object into your application by using the OLE container control or by adding the object's class to the Toolbox.

To fully understand ActiveX components, you should first be familiar with how to work with classes, objects, properties, and methods, which are explained in Chapter 9, "Programming with Objects."

Contents

Sample Applications: Geofacts.vbp and Olecont.vbp

Many of the concepts in this chapter are demonstrated in the sample applications Geofacts.vbp and Olecont.vbp. If you installed the sample applications, you will find the Geofacts.vbp application in the \Geofacts subdirectory and the Olecont.vbp application in the \Olecont subdirectory of the Visual Basic Programmer's Guide samples directory (\VB\Samples\PGuide).

Types of ActiveX Components

ActiveX components give you the power to put together sophisticated applications from pieces that already exist. Your Visual Basic applications can include several types of ActiveX components:

- Applications that support ActiveX technology, such as Microsoft Excel, Microsoft Word, and Microsoft Access, provide objects that you can manipulate programmatically from within your Visual Basic application. For example, you can use the properties, methods, and events of a Microsoft Excel spreadsheet, Microsoft Word document, or Microsoft Access database in your application.
- Code components provide libraries of programmable objects. For example, a code component could include a library of specialized financial functions for spreadsheet users, or user-interface elements, such as dialog boxes, that are common to multiple applications. Unlike an object in an ActiveX-enabled application, an object in a code component can run in the same process as your application, allowing faster access to the object.
- You can add features without having to create them yourself by using ActiveX controls as components. ActiveX controls are available from a variety of vendors to provide many specialized features, such as displaying a calendar on a form or reading data in a particular format.
- ActiveX documents let you create interactive Internet applications. You can create forms that can be contained within Internet Explorer. ActiveX documents can show message boxes and secondary forms and contain ActiveX controls. ActiveX documents can also function as code components. For a step-by-step introduction to ActiveX documents, see "Creating an ActiveX Document" in the *Component Tools Guide,* available in the Professional and Enterprise editions.

Some ActiveX components run in the same process as your application, while others run in a separate process. For more information, see "In-Process and Out-of-Process Servers."

In addition to components in existing ActiveX-enabled applications, code component libraries, ActiveX controls, and ActiveX documents, you can create your own components. For more information on creating your own ActiveX components, see "Creating ActiveX Components" in the *Component Tools Guide,* available in the Professional and Enterprise editions.

In-Process and Out-of-Process Servers

ActiveX components interact with your application and with each other through a *client/server* relationship. The *client* is the application code or component that uses the features of a component. The *server* is the component and its associated objects. For example, suppose your application uses an ActiveX control to provide a standard Employee form for multiple applications in your company. The ActiveX control that provides the Employee form is the server; the applications that use the control are its clients.

Depending on how an ActiveX component has been implemented, it may run in the same process as its client applications, or in a different process. For example, if your application uses a component that is part of an ActiveX-enabled application, it runs in a separate process. If the component has been implemented as a programmable object in a dynamic-link library (.dll file), it runs in the same process as your application.

In general, if an ActiveX component has been implemented as part of an executable file (.exe file), it is an *out-of-process* server and runs in its own process. If it has been implemented as a dynamic-link library, it is an *in-process* server and runs in the same process as the client application. Applications that use in-process servers usually run faster than those that use out-of-process servers because the application doesn't have to cross process boundaries to use an object's properties, methods, and events.

The following table shows how you can implement the different types of components:

Component	**Server Type**
ActiveX-enabled application	Out-of-process
Code component	Either in-process or out-of-process

ActiveX control	In-process
ActiveX document	Either in-process or out-of-process

Using in-process components is one way to optimize the performance of your application. Another way to optimize performance is to use early binding. For more information, see "Speeding Object References" later in this chapter.

Working with ActiveX Components

You work with object provided by ActiveX components in much the same way that you work with other objects. You assign an object reference to a variable, then write code that uses the object's methods, properties, and events. However, there are some things you need to be aware of when you work with objects provided by components.

This topic provides an overview of the top-level tasks for working with objects provided by components and an example of using objects in an ActiveX-enabled application. For details on each task, see the appropriate topic described under each task item.

To use most objects provided by ActiveX components

1. Create a reference to the object you want to use. How you do this depends on the type of object and whether the ActiveX component supplies a type library.
2. For more information, see "Creating a Reference to an Object" later in this chapter.
3. Write code using the object's methods, properties, and events.
4. For more information, see "Using an Object's Properties, Methods, and Events" later in this chapter.
5. Release the object when you are finished using it.
6. For more information, see "Releasing an ActiveX Component" later in this chapter.
7. Create error-handlers; see "Handling Run-Time Errors in ActiveX Components" later in this chapter.

For example, suppose you have created a form with three text boxes (Text1, Text2, and Text3) and a command button (Command1), and added a reference in your project to the Microsoft Excel 8.0 Object Library. You can then add code to the command button's Command1_Click event procedure that uses the Microsoft Excel Formula method to add two numbers entered in Text1 and Text2, displaying the result in Text3. (To avoid a type mismatch error, you may want to remove the default text value of each text box by setting its Text property to an empty string):

```
Private Sub Command1_Click()
   ' Declare object variables for Microsoft Excel,
   ' application workbook, and worksheet objects.
   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet

   ' Assign object references to the variables. Use
   ' Add methods to create new workbook and worksheet
   ' objects.
   Set xlApp = New Excel.Application
   Set xlBook = xlApp.Workbooks.Add
   Set xlSheet = xlBook.Worksheets.Add

   ' Assign the values entered in the text boxes to
   ' Microsoft Excel cells.
   xlSheet.Cells(1, 1).Value = Text1.Text
   xlSheet.Cells(2, 1).Value = Text2.Text

   ' Use the Formula method to add the values in
   ' Microsoft Excel.
   xlSheet.Cells(3, 1).Formula = "=R1C1 + R2C1"
```

```
    Text3.Text = xlSheet.Cells(3, 1)

    ' Save the Worksheet.
    xlSheet.SaveAs "c:\Temp.xls"

    ' Close Microsoft Excel with the Quit method.
    xlApp.Quit

    ' Release the objects.
    Set xlApp = Nothing
    Set xlBook = Nothing
    Set xlSheet = Nothing
End Sub
```

For simplicity, this example doesn't include error handling. However, it is highly recommended that you include error handling in applications that use objects provided by ActiveX components.

Creating a Reference to an Object

Before you can use an object's properties, methods, and events in your application, you must declare an object variable, then assign an object reference to the variable. How you assign an object reference depends on two factors:

- Whether the ActiveX component supplies a type library. An ActiveX component's type library contains definitions of all the objects the component provides, including definitions for all available methods, properties, and events. If an ActiveX component provides a type library, you need to add a reference to the type library in your Visual Basic project before you can use the library's objects.
- Whether the object is a top-level, *externally creatable object*, or a *dependent object.* You can assign a reference to an externally created object directly, while references to dependent objects are assigned indirectly.

If an object is externally creatable, you can assign an object reference to a variable by using the New keyword, CreateObject, or GetObject in a Set statement from outside the component. If the object is a dependent object, you assign an object reference by using a method of a higher-level object in a Set statement.

In Microsoft Excel, for example, an Application object is an externally creatable object you can assign a reference to it directly from your Visual Basic application by using the New keyword, CreateObject, or GetObject in a Set statement. A Range object, by contrast, is a dependent object you assign a reference to it by using the Cells method of a Worksheet object in a Set statement. For more information on externally creatable and dependent objects, see "Navigating Object Models" later in this chapter.

If the object's class is included in a type library, you can make your application run faster by creating an object reference using a variable of that specific class. Otherwise, you must use a variable of the generic Object class, which results in late binding. For more information, see "Speeding Object References."

To create a reference to an object defined in a type library

1. From the **Project** menu, choose **References**.
2. In the **References** dialog box, select the name of the ActiveX component containing the objects you want to use in your application.
3. You can use the **Browse** button to search for the type library file containing the object you need. Type libraries can have a .tlb or .olb file-name extension. Executable (.exe) files and dynamic link libraries (dlls) can also supply type libraries, so you can also search for files with these file-name extensions.
4. If you are not sure if an application is ActiveX-enabled and supplies a type library, try adding a reference to it using the **Browse** button. If the reference fails, Visual Basic displays the error message, "Can't add a reference to the specified file," indicating that the type library doesn't exist. For more information about working with objects that aren't associated with a type library, see "Creating a Reference to an Object."
5. From the **View** menu, choose **Object Browser** to view the referenced type library. Select the appropriate type library

from the **Project/Library** list. You can use all the objects, methods, and properties listed in the **Object Browser** in your application.

6. For more information on using the Object Browser, see "Browsing ActiveX Component Type Libraries."
7. Declare an object variable of the object's class. For example, you could declare a variable of the class Excel.Chart to refer to a Microsoft Excel Chart object.

```
Dim xlChart As Excel.Chart
```

8. For more information, see "Declaring an Object Variable" later in this chapter.
9. Assign an object reference to the variable by using the New keyword, CreateObject, or GetObject in a Set statement. For more information, see "Assigning an Object Reference to a Variable" later in this chapter.
10. If the object is a dependent object, assign an object reference by using a method of a higher-level object in a Set statement.

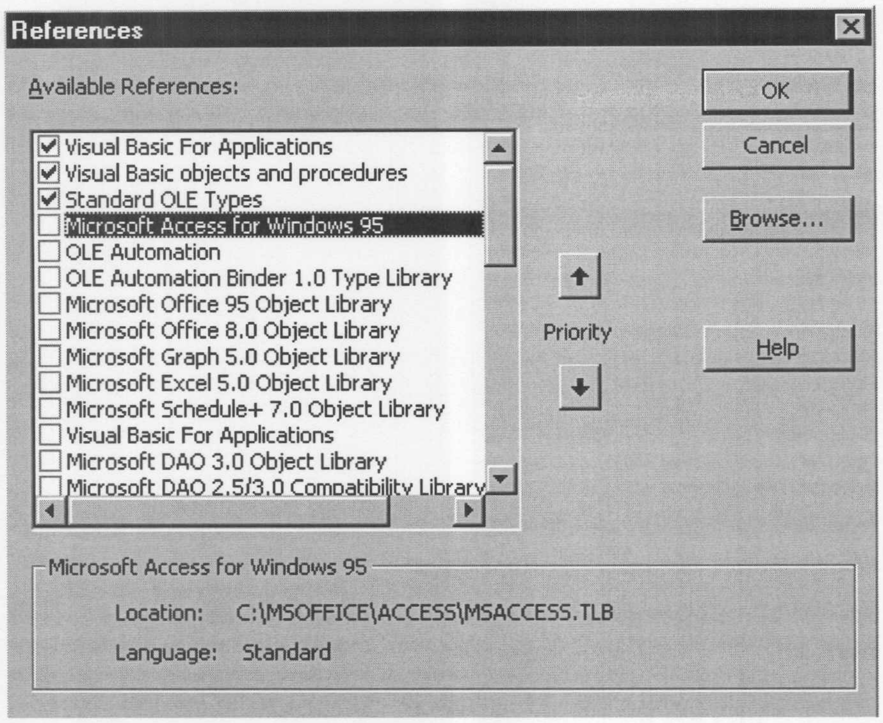
To create a reference to an object not defined in a type library

1. Declare an object variable of the Object data type.
2. Because the object isn't associated with a type library, you won't be able to use the **Object Browser** to view the properties, methods, and events of the object. You need to know what properties, methods, and events the object provides, including any methods for creating a reference to a dependent object.
3. For more information, see "Declaring an Object Variable" later in this chapter.
4. Assign an object reference to the variable by using CreateObject or GetObject in a Set statement. For more information, see "Assigning an Object Reference to a Variable" later in this chapter.
5. If the object is a dependent object, assign an object reference by using a method of a higher-level object in a Set statement.

Ambiguous References and Reference Priority

When you refer to a constant or object in code, Visual Basic searches for the constant or object class in each type library selected in the References dialog box in the order the type libraries are displayed. If two type libraries contain constants or classes with identical names, Visual Basic uses the definition provided by the type library listed higher in the Available References box.

Figure 10.1 The References dialog box



☑ Visual Basic For Applications
☑ Visual Basic objects and procedures
☑ Standard OLE Types
☐ Microsoft Access for Windows 95
☐ OLE Automation
☐ OLE Automation Binder 1.0 Type Library
☐ Microsoft Office 95 Object Library
☐ Microsoft Office 8.0 Object Library
☐ Microsoft Graph 5.0 Object Library
☐ Microsoft Excel 5.0 Object Library
☐ Microsoft Schedule+ 7.0 Object Library
☐ Visual Basic For Applications
☐ Microsoft DAO 3.0 Object Library
☐ Microsoft DAO 2.5/3.0 Compatibility Library

Microsoft Access for Windows 95
Location: C:\MSOFFICE\ACCESS\MSACCESS.TLB
Language: Standard

The best way to handle potentially ambiguous references is to explicitly specify the type library that supplies the constant or class when you use it. For example, the constant vbCancel evaluates to different values in the Visual Basic and Visual Basic for Applications type libraries. The following code shows fully qualified and ambiguous references to the constant vbCancel:

```
' Print the Visual Basic vbCancel.
Debug.Print "VB.vbCancel = "; VB.vbCancel
' Print the Visual Basic for Applications vbCancel.
Debug.Print "VBA.vbCancel = "; VBA.vbCancel
' Ambiguous reference prints the value of vbCancel
' that appears highest in the type library highest
' in the Available References list.
Debug.Print "vbCancel = "; vbCancel
```

The following code example shows fully qualified and ambiguous declarations for an Application object variable. If Microsoft Word appears higher in the Available References box than Microsoft Excel, xlApp2 is declared using the Microsoft Word Application class rather than the Microsoft Excel Application class.

```
' Fully qualified object variable declaration.
Dim xlApp1 As Excel.Application
' Ambiguous object variable declaration.
Dim xlApp2 As Application

' Assign an object reference.
Set xlApp1 = New Excel.Application
' The following generates a type mismatch error.
Set xlApp2 = xlApp1
```

You may be tempted to handle potentially ambiguous references by changing the order in which Visual Basic searches for references. The References dialog box includes two Priority buttons that let you move a type library higher in the list, so that its constants and classes will be found sooner than constants or classes with identical names lower on the list. However, changing the priority order can cause unexpected problems in your applications if there are other ambiguous references. In general, it's better to explicitly specify the type library in any references.

Note The Excel.Application syntax for referring to the Microsoft Excel Application class is not supported in versions prior to Microsoft Excel 97. To refer to the Microsoft Excel Application class in Microsoft Excel 5.0 and Microsoft Excel 95, use the syntax [_ExcelApplication] instead. For example:

```
Set xlApp = New [_ExcelApplication]
```

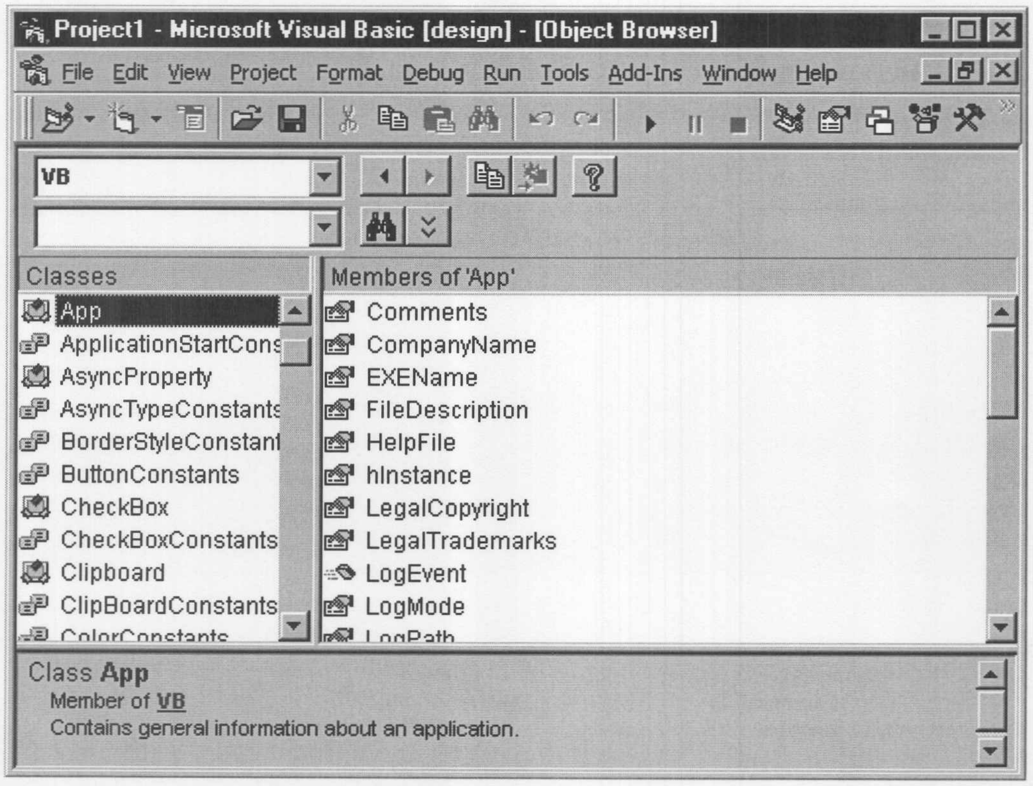
Browsing ActiveX Component Type Libraries

If an ActiveX component provides a type library, you can use the Object Browser to view the component's classes, as well as the properties, methods, events, and constants associated with the objects of each class.

To view the classes available in an ActiveX Component's type library

1. If you haven't already done so, add a reference to the type library to your Visual Basic project.
2. For more information, see "Creating a Reference to an Object," later in this chapter.
3. Open the **Object Browser** and select the name of the type library from the Project/Library list.
4. The **Object Browser** displays the available classes in the **Classes** list.

Figure 10.2 The Object Browser



Class **App**
Member of **VB**
Contains general information about an application.

For example, to view the classes available in the Data Access Object (DAO) type library, add a reference to the library in the References dialog box, then select DAO in the Project/Library list in the Object Browser.

To view the members of a class

- Select the name of the class from the Classes list in the **Object Browser**.

- The **Object Browser** displays the members of the class in the **Members of** list.

If you're looking for information about a particular class or member in a type library, use the Object Browser's Search feature.

To use the Search feature

- Type what you're looking for in the Search Text box, and then click the **Search** button.
- The **Object Browser** displays a Search Results box showing the libraries, classes, and members returned by the search.

Declaring an Object Variable

Before you can use the properties, methods, and events of an object provided by an ActiveX component, you must first declare an object variable. The way you declare an object variable depends on whether or not the ActiveX component supplies a type library.

To declare a variable for an object defined in a type library

1. Add a reference to the type library to your Visual Basic project. For more information on adding a reference to a type library, see "Creating a Reference to an Object" earlier in this chapter.
2. Specify the name of a class supplied by that type library in your variable declaration. Declaring an object variable of a specific class can speed object references. Use the following syntax:

   Dim *variable* **As** [**New**] *class*

3. The *class* argument can be composed of two parts, in the form *component.class*.

Part	Description
component	The name of the component that supplies the object. Choices are displayed in the Project/Library list of the Object Browser.
class	The object's class name (provided by the component's type library). Choices are shown in the Classes/Modules box of the Object Browser.

For example, you can declare a variable for a Microsoft Excel Chart object in either of the following ways:

```
Dim xlChart As Chart
Dim xlChart As Excel.Chart
```

If you declare an object variable using the New keyword, Visual Basic will automatically create an object and assign an object reference the first time you use the variable. For example, the following statements assign a reference to a new DAO table object to the variable tdfOrders, setting the table's Name property to "Orders":

```
Dim tdfOrders As New TableDef
tdfOrders.Name = "Orders"
```

Note Using variables declared using the New keyword can slow your application. Every time Visual Basic encounters a variable declared using New, it must test whether or not an object reference has already been assigned to the variable.

To declare an object variable for an object not defined in a type library

- Declare an object variable of the generic Object class, as follows:

  Dim *variable* **As Object**

For example, the variable **objAny** in the following declaration can be used for a Microsoft Excel Chart object or any other object provided by an ActiveX component:

```
Dim objAny As Object
```

The main difference between declaring a variable of a specific class and declaring a variable of the generic Object class is in how ActiveX binds the variable to the object. When you declare a variable of the generic Object class, ActiveX must use late binding. When you declare an object variable of a specific class, ActiveX uses early binding, which can speed object references. For more information, see "Speeding Object References" later in this chapter.

For More Information For more information on declaring object variables, see "Dim Statement" in the *Language Reference* in Books Online. For more information on assigning an object reference to a variable, see "Assigning an Object Reference to a Variable."

Assigning an Object Reference to a Variable

After you declare an object variable, you must assign an object reference to the variable before you can use the object's properties, methods, and events. You can assign a new object reference in several ways:

- If you declared the variable using the New keyword, Visual Basic will automatically assign a new object reference the first time you use the variable.
- You can assign a reference to a new object in a Set statement by using the New keyword or CreateObject function.
- You can assign a reference to a new or existing object in a Set statement by using the GetObject function.

Assigning an Object Reference Using the New Keyword

If the ActiveX component supplies a type library, you can use the New keyword in a variable declaration or Set statement to create a new object and assign an object reference to an object variable.

If you declare an object variable with the New keyword, Visual Basic will automatically create a new object the first time you use the variable. For more information, see "Declaring an Object Variable."

You can also use the New keyword in a Set statement to assign a reference to a new object of the specified class. For example, the following statements assign a reference to a new DAO table object to the variable tdfOrders, setting the table's Name property to "Orders":

```
Dim tdfOrders As DAO.TableDef
Set tdfOrders = New DAO.TableDef
tdfOrders.Name = "Orders"
```

For More Information See "Dim Statement" or "Set Statement" in the *Language Reference* in Books Online.

Assigning an Object Reference Using CreateObject

Regardless of whether or not an ActiveX component supplies a type library, you can use the CreateObject function in a Set statement to create a new object and assign an object reference to an object variable. You must specify the object's programmatic identifier as an argument to the function, and the object you want to access must be externally creatable.

To assign an object reference using CreateObject

- Use the following syntax for CreateObject.
  Set *objectvariable* = **CreateObject("***progID***")**

The *progID* argument is usually the fully qualified class name of the object being created; for example, Word.Document. However, progID can be different from the class name. For example, the progID for a Microsoft Excel object is "Sheet" rather than "Worksheet."

The following code example starts Microsoft Excel (if Microsoft Excel is not already running) and establishes the variable **xlApp** to refer to an object of the Application class. The argument **"Excel.Application"** fully qualifies Application as a class defined by Microsoft Excel:

```
Dim xlApp As Excel.Application
Set xlApp = CreateObject("Excel.Application")
```

For More Information See "CreateObject Function" in the *Language Reference* in Books Online.

Assigning an Object Reference Using GetObject

The GetObject function is most often used to assign a reference to an existing object, although you can also use it to assign a reference to a new object.

To assign a reference to an existing object, use the following syntax.

Set *objectvariable* = **GetObject([***pathname***] [,** *progID***])**

The *pathname* argument can be the path to an existing file, an empty string, or omitted entirely. If it is omitted, then *progID* is required. Specifying the path to an existing file causes GetObject to create an object using the information stored in the file. Using an empty string for the first argument causes GetObject to act like CreateObject — it will create a new object of the class whose programmatic identifier is progID. The following table describes the results of using GetObject.

If the ActiveX component is running	Result
Set X = GetObject(, "MySrvr.Application")	X references an existing Application object.
Set X = GetObject("", "MySrvr.Object")	X references a new, externally creatable object.

If the ActiveX component is not running	Result
Set X = GetObject(, "MySrvr.Object")	An error is returned.

Set X = GetObject("", "MySrvr.Object")	The ActiveX component (MySrvr) is started, and X references a new object.

For example, the variable **wrdApp** refers to a running Microsoft Word Application:

```
Dim wdApp As Word.Application
Set wdApp = GetObject("", "Word.Application")
```

Just as with CreateObject, the argument **"Word.Application"** is the programmatic identifier for the Application class defined by Microsoft Word. If multiple instances of Microsoft Word are running, you cannot predict to which instance **wdApp** will refer.

Important You can also use GetObject to assign a reference to an object in a compound document file. A *compound document file* contains references to multiple types of objects. For example, a compound document file could contain a spreadsheet, text, and bitmaps.

The following example starts the spreadsheet application, if it is not already running, and opens the file Revenue.xls:

```
Dim xlBook As Excel.Workbook
Set xlBook = GetObject("C:\Accounts\Revenue.xls")
```

For More Information See "GetObject Function" in the *Language Reference* in Books Online.

Speeding Object References

You can make your Visual Basic applications run faster by optimizing the way Visual Basic resolves object references. The speed with which Visual Basic handles object references can be affected by:

- Whether or not the ActiveX component has been implemented as an in-process server or an out-of-process server.
- Whether an object reference is early-bound or late-bound.

In general, if a component has been implemented as part of an executable file (.exe file), it is an *out-of-process* server and runs in its own process. If it has been implemented as a dynamic-link library, it is an *in-process* server and runs in the same process as the client application.

Applications that use in-process servers usually run faster than those that use out-of-process servers because the application doesn't have to cross process boundaries to use an object's properties, methods, and events. For more information about in-process and out-of-process servers, see "In-Process and Out-of-Process Servers."

Object references are *early-bound* if they use object variables declared as variables of a specific class. Object references are *late-bound* if they use object variables declared as variables of the generic Object class. Object references that use early-bound variables usually run faster than those that use late-bound variables.

For example, you could assign a reference to an Excel object to either of the following variables:

```
Dim xlApp1 As Excel.Application
Set xlApp1 = New Excel.Application
```

```
    Dim xlApp2 As Object
    Set xlApp2 = CreateObject("Excel.Application")
```

Code that uses variable **xlApp1** is early-bound and will execute faster than code that uses variable **xlApp2**, which is late-bound.

Late Binding

When you declare a variable As Object, Visual Basic cannot determine at compile time what sort of object reference the variable will contain. In this situation, Visual Basic must use *late binding*— that is, Visual Basic must determine at run time whether or not that object will actually have the properties and methods you used in your code.

For example, Visual Basic will compile the following code without generating errors, even though it refers to a method that doesn't exist, because it uses a late-bound object variable. It doesn't check for the existence of the method until run time, so it will produce a run-time error:

```
    Dim xlApp As Object
    Set xlApp = CreateObject("Excel.Application")
    xlApp.TheImpossibleMethod     ' Method doesn't exist.
```

This code runs slower than code that uses an early-bound object variable because Visual Basic must include code in the compiled executable that will determine at run time whether or not the Microsoft Excel Application object has a TheImpossibleMethod method.

Although late binding is the slowest way to invoke the properties and methods of an object, there are times when it is necessary. For example, you may write a function that uses an object variable to act on any of several different classes of objects. Because you don't know in advance what class of object will be assigned to the variable, declare it as a late-bound variable using As Object.

Early Binding

If Visual Basic can detect at compile time what object a property or method belongs to, it can resolve the reference to the object at compile time. The compiled executable contains only the code to invoke the object's properties, methods, and events. This is called *early binding*.

When you declare an object variable using the class that defines the object, the variable can only contain a reference to an object of that class. Visual Basic can use early binding for any code that uses the variable.

Early binding dramatically reduces the time required to set or retrieve a property value, because the call overhead can be a significant part of the total time. For method calls, the improvement depends on the amount of work the method does. Short methods, where the call overhead is comparable to the time required to complete the task, will benefit the most.

Using an Object's Properties, Methods, and Events

After you assign an object reference to an object variable, you can use the variable to manipulate the object's properties and methods. You can also declare an object variable using the WithEvents keyword and use it to make your application respond to the object's events.

Using an Object's Properties and Methods

You can use the *object.property* syntax to set and return an object's property values or the *object.method* syntax to use methods on the object. For example, you could set the Caption property of the Application object as follows:

```
Dim xlApp As Excel.Application
Set xlApp = New Excel.Application
xlApp.Caption = "MyFirstObject"
```

Note The Excel.Application syntax for referring to the Microsoft Excel Application class is not supported in versions prior to Microsoft Excel 97. To refer to the Microsoft Excel Application class in Microsoft Excel 5.0 and Microsoft Excel 95, use the syntax [_ExcelApplication] instead. For example:

```
Set xlApp = New [_ExcelApplication]
```

You could call the Quit method of the Microsoft Excel Application object like this:

```
xlApp.Quit
```

You could set the Caption property of the Application object like this:

```
xlApp.Caption = "MyFirstObject"
```

In general, it is a good idea to be as specific as possible when referring to methods or properties of objects defined by other applications or projects. For example:

```
' Fully qualified property name sets
' the Microsoft Project window caption.
Dim pjWindow As Project.Window
' Get a reference to the first Window object.
Set pjWindow = ActiveProject.Windows(1)
pjWindow.Caption = "Project Caption"

' Unqualified name causes Visual Basic to use
' the first object it finds with a property
' named Caption - in this case, Form1.
Caption = "Microsoft Form1 Caption"
```

Note If you need to import binary data into your Visual Basic application and you plan to share the data between applications using ActiveX, use a Byte array to store the data. If you assign binary data to a string and then try to pass this data to an Automation object that takes a string, the data may not be converted correctly. For more information on data types, see Chapter 5, "Programming Fundamentals.

For More Information For more information on working with an object's properties and methods, see Chapter 9, "Programming with Objects.

Responding to an Object's Events

In addition to responding to events that occur to Visual Basic objects, your application can respond to events in an object provided by an ActiveX component. For example, your Visual Basic application can display a message box if an event occurs in a Microsoft Excel workbook.

You make your application respond to an object's events by adding code to an event procedure for the object. However, event procedures for objects provided by components are not automatically available in Visual Basic. You must first declare an object variable using the WithEvents keyword.

After you declare an object variable using WithEvents, the Visual Basic code window uses the variable to display event procedures for the object. You can then add code to these event procedures to respond to the object's events. When you assign an object reference to the variable, you establish a connection between the variable and the object at run time.

To create an event procedure for an object provided by a component

1. Add a reference to the component's type library to your Visual Basic project. For more information on adding a reference to a type library, see "Creating a Reference to an Object."
2. In the Declarations section of a standard module, declare an object variable using the WithEvents keyword. For example:

```
Dim WithEvents xlBook As Excel.Workbook
```

3. Visual Basic adds the name of the object variable to the Object box in the code window. When you select the variable name, Visual Basic displays the object's event procedures in the Procedure list box.
4. Select an event procedure, then add code to the procedure that you want your application to run when the event occurs.
5. For example, suppose your Visual Basic application relies on data displayed in a Microsoft Excel workbook and that you've already declared a WithEvents variable xlBook for the workbook. When a user tries to close the workbook, you can display a message and keep the workbook from closing by adding the following code to the xlBook_BeforeClose event procedure in your application:

```
    Private Sub xlBook_BeforeClose(Cancel As Boolean)
        ' Hide the Microsoft Excel window so the message
        ' will be visible.
        xlBook.Application.Visible = False
        ' Display the message.
        MsgBox "This workbook must remain open."
        ' Unhide the Microsoft Excel window.
        xlBook.Application.Visible=True
        ' Set the event procedure's Cancel argument
        ' to True, cancelling the event.
        Cancel = True
    End Sub
```

6. Assign an object reference to the WithEvents object variable.
7. For example, you could add the following to the Visual Basic form's Form_Load event procedure to assign the variable xlBook a reference to a Microsoft Excel workbook, Sales.xls: Private Sub Form_Load() Set xlBook = GetObject("Sales.xls") ' Display Microsoft Excel and the Worksheet window. xlBook.Application.Visible = True xlBook.Windows(1).Visible = True End Sub

For More Information See "Dim Statement" in the *Language Reference* in Books Online.

Releasing an ActiveX Component

When you are finished using an object, clear any variables that reference the object so the object can be released from memory. To clear an object variable, set it to Nothing. For example:

```
Dim acApp As Access.Application
Set acApp = New Access.Application
MsgBox acApp.SysCmd(acSysCmdAccessVer)
Set acApp = Nothing
```

All object variables are automatically cleared when they go out of scope. If you want the variable to retain its value across procedures, use a public or form-level variable, or create procedures that return the object. The following code shows how you would use a public variable:

```
  Public wdAppas Word.Application
.
.
.
  ' Create a Word object and start Microsoft Word.
  Set wdApp = New Word.Application
.
.
.
  ' Microsoft Word will not close until the
  ' application ends or the reference is set to Nothing:
  Set wdApp = Nothing
```

Also, be careful to set all object references to Nothing when finished, even for dependent objects. For example:

```
  Dim xlApp As Excel.Application
  Dim xlBook As Excel.Workbook
  Set xlApp = New Excel.Application
  Set xlBook = xlApp.Workbooks.Add
  Set xlApp = Nothing       ' Careful! xlBook may still
                            ' contain an object reference.
  Set xlBook = Nothing      ' Now all the references
                            ' are cleared.
```

Navigating Object Models

Once you understand how to use objects provided by components, you can use any object that is a component exposes to you. Components can range from a simple code component or ActiveX control to large components, such as Microsoft Excel and the Microsoft Data Access Object (DAO) programming interface, which expose many objects.

Each object exists somewhere in the component's object hierarchy, and you can access the objects in two ways:

- Directly, if the object is externally creatable.
- Indirectly, if the object is a dependent object. You can get a reference to it from another object higher in the component's hierarchy.

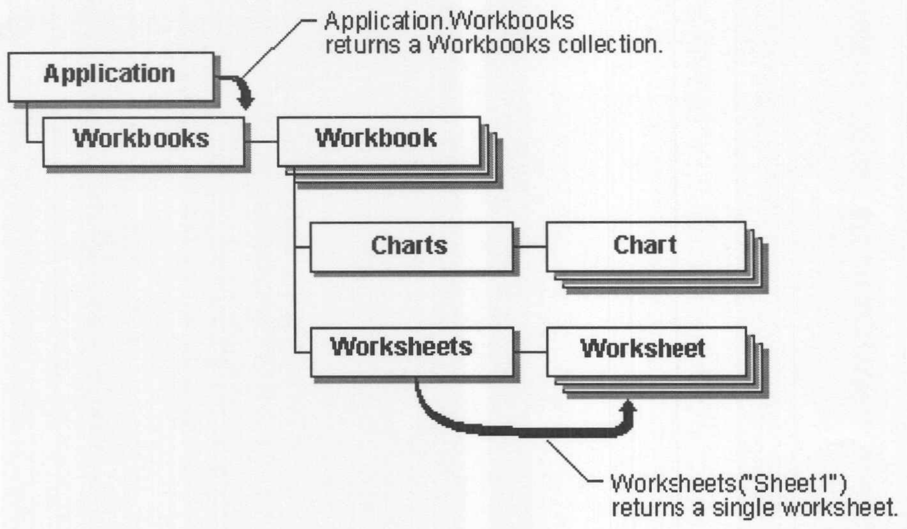
The best way to navigate an object hierarchy is to use the Object Browser (if the component provides an object library).

Navigating the Object Hierarchy

As you've seen, you navigate down an object hierarchy by setting references to dependent objects through externally creatable objects. You can also use a method on a collection object to return an individual object. For more information see "Working with Externally Creatable and Dependent Objects."

Figure 10.3 shows the object navigation path in a Microsoft Excel application.

Figure 10.3 Navigating down a Microsoft Excel object hierarchy using collections



Collection Objects

Collection objects are containers for groups of other objects. These objects provide an easy way to keep track of a set of objects that are of the same type. For example, a collection of all the Menu objects in an application can be accessed using the Menus collection object. You can use the following code to refer to *all* the workbooks that are currently loaded in Microsoft Excel:

Application.Workbooks

Notice that Workbooks is plural. The standard naming convention for collection objects is the plural of the type of object that makes up the collection. You can iterate through the objects in a collection by using the For Each statement, as follows:

```
Dim xlBook As Excel.Workbook
   .
   .
   .
For Each xlBook In Application.Workbooks
   ' Display the name of each workbook.
   MsgBox xlBook.FullName
Next xlBook
```

Individual objects in many collections can also be referenced by name or by their index order in the collection. The following example shows how you would refer to Style objects named "Normal," "Example," and "Heading":

```
xlBook.Styles("Normal")
xlBook.Styles("Example")
xlBook.Styles("Heading")
```

Assuming these objects are the first three objects in the Styles, and that the collection is zero-based, you could also refer to them as follows:

```
xlBook.Styles(1)     ' Refers the Normal Style object.
xlBook.Styles(2)     ' Refers the Example Style object.
xlBook.Styles(3)     ' Refers the Heading Style object.
```

For More Information For more information on working with collection objects, see Chapter 9, "Programming with Objects."

Working with Externally Creatable and Dependent Objects

How you create a reference to an object provided by a component depends on whether the object is an externally creatable or dependent object. You can directly create a reference to an externally creatable object; you create a reference to a dependent object indirectly by using a method of a higher-level object in the component's object hierarchy.

Externally Creatable Objects

Most large ActiveX-enabled applications and other ActiveX components provide a top-level externally creatable object in their object hierarchy that:

- Provides access to other objects in the hierarchy.
- Provides methods and properties that affect the entire application.

For example, the Microsoft Office applications each provide a top-level Application object. The following example shows how you can assign references to the Application objects of Microsoft Excel, Microsoft Word, and Microsoft Access:

```
Dim xlApp As Excel.Application
Dim wdApp As Word.Application
Dim acApp As Access.Application

Set xlApp = New Excel.Application
Set wdApp = New Word.Application
Set acApp = New Access.Application
```

You can then using these variables to access the dependent objects in each application and the properties and methods of these objects. For more information see "Creating a Reference to an Object."

Note The Excel.Application syntax for referring to the Microsoft Excel Application class is not supported in versions prior to Microsoft Excel 97. To refer to the Microsoft Excel Application class in Microsoft Excel 5.0 and Microsoft Excel 95, use the syntax [_ExcelApplication] instead. For example:

```
Set xlApp = New [_ExcelApplication]
```

In addition to these top-level externally creatable objects, ActiveX components can also provide externally creatable objects that are lower on the component's object hierarchy. You can access these objects either directly as an externally creatable object or indirectly as a dependent object of a higher-level externally creatable object. For example, you can create a reference to a DAO TableDef object either directly or indirectly:

```
' Create a reference to daoTable1 directly.
Dim daoTable1 As DAO.TableDef
Set daoTable1 = New DAO.TableDef
daoTable1.Name = "Table1"

' Create a reference to daoTable2 indirectly,
' as a dependent object of the DAO DBEngine object.
Dim daoDBE As DAO.DBEngine
```

```
    Dim daoWs As DAO.Workspace
    Dim daoDb As DAO.Database
    Dim daoTable2 As DAO.TableDef

    Set daoDBE = DAO.DBEngine
    Set daoWs = daoDBE.Workspaces(0)
    Set daoDb = daoWs.CreateDatabase("db1.mdb", dbLangGeneral)
    Set daoTable2 = daoDb.CreateTableDef("Table2")
```

Some objects provide an Application object, but give it a different name. For example, the Microsoft Jet database engine in Microsoft Access calls its top-level object the DBEngine object.

Dependent Objects

You can get a reference to a dependent object in only one way — by using a property or method of an externally creatable object to return a reference to the dependent object. Dependent objects are lower in an object hierarchy, and they can be accessed only by using a method of an externally creatable object. For example, suppose you want a reference to a Button object from Microsoft Excel. You can't get a reference to this object using the following code (an error will result):

```
Dim xlButton As Excel.Button
Set xlButton = New Excel.Button
```

Instead, use the following code to get a reference to a Button object:

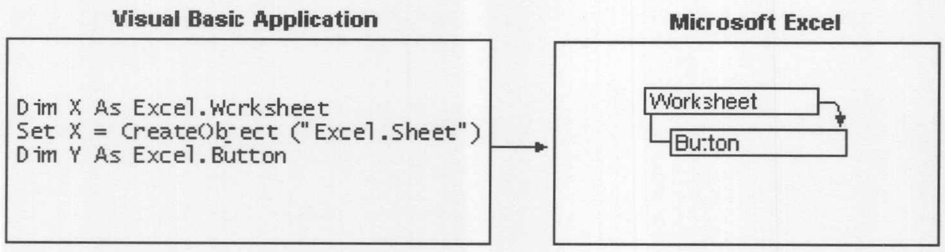
```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.Worksheet
Dim xlButton As Excel.Button

Set xlApp = New Excel.Application
Set xlBook = xlApp.Workbooks.Add
Set xlSheet = xlBook.Worksheets.Add
Set xlButton = xlSheet.Buttons.Add(44, 100, 100, 44)

' Now you can use a Button object property.
xlButton.Caption = "FirstButton"
```

Figure 10.5 illustrates how a Visual Basic application gets a reference to the Button object.

Figure 10.5 Accessing dependent objects



```
Dim X As Excel.Worksheet
Set X = CreateObject ("Excel.Sheet")
Dim Y As Excel.Button
```

Handling Run-Time Errors in ActiveX Components

Error-handling code is especially important when you're working with ActiveX components, because code from the component is used from within your Visual Basic application. Where possible, you should include code to handle errors that the component may generate. For example, it is good practice to check for the error that occurs if a user unexpectedly closes a component application:

Function StartWord() ' Starts Microsoft Word. On Error Goto ErrorTrap ' Declare a Microsoft Word Application variable ' and an integer variable for error trap. Dim wdApp As Word.Application Dim iTries As Integer ' Assign an object reference. Set wdApp = New Word.Application ' Release object variable. Set wdApp = Nothing Exit Function ErrorTrap: ' Trap for the error that occurs if Microsoft Word ' can't be started. Select Case Err.Number Case 440 ' Automation error. iTries = iTries + 1 ' Make up to 5 attempts to restart Word. If iTries < 5 Then Set wdApp = New Word.Application Resume Else Err.Raise Number:=VBObjectError + 28765, _ Description:= "Couldn't restart Word" End If Case Else Err.Raise Number:= Err.Number End Select End Function

If any error other than error 440 occurs in the preceding example, the procedure displays the error and raises an error. The application that provides the object might pass back its own error. In some cases, an application might use the same error code that Visual Basic uses for a different error. In these cases, you should use On Error Resume Next and check for errors immediately after each line that might cause an error. This type of error checking is called *inline error-handling.*

Testing for Object References

Before using an object variable in your code, you may want to make sure the variable holds a valid object reference. You can determine whether or not an object reference has been assigned to the variable by using Is Nothing. For example, the following code checks whether or not an object reference has been assigned to the variable wdDoc:

```
If wdDoc Is Nothing Then MsgBox "No object reference."
```

However, Is Nothing won't detect whether or not a valid object reference has become unavailable. For example, if you assign a Microsoft Word object reference to an object variable and Microsoft Word becomes unavailable, the variable will still hold a valid object reference. In this situation, use your error handler to trap the error that results when your code tries to use the variable.

For More Information For information about the errors that a particular application might pass back, see that application's documentation. For more information about trapping errors, see Chapter 13 "Debugging Your Code and Handling Errors."

Handling Requests Pending to an ActiveX Component

It usually takes only a fraction of a second to set a property or call a method of an object. At times, however, your request may not be finished immediately.

- If you call the Close method of a Workbook in Microsoft Excel while the user has a dialog box open, Microsoft Excel signals that it is busy and cannot execute your request. This can lead to a *component busy* condition.
- If you call a method that performs a lengthy operation, such as a large amount of database work when the database is very active, you may try to perform another operation while the first operation is still pending. This can lead to a *request pending* condition.
- If you have two or more programs making calls to a shared component, one call must be completed before another can begin. Components handle such conflicts by *serializing* the requests, that is, making them wait in line. This can also lead to a request pending condition.

Component busy is like getting a busy signal when you make a telephone call. You know you're not going to get through, so you may as well hang up and try again later.

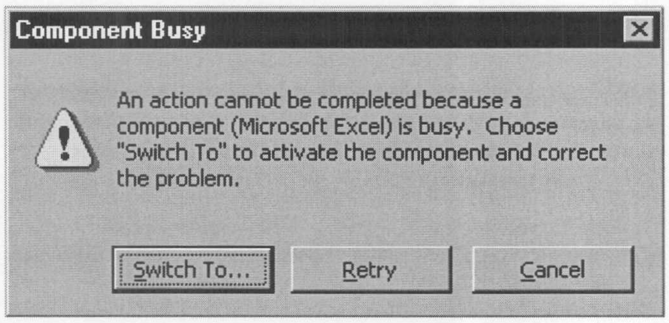
Request pending is like having your call go through, and then having the person you're calling keep you talking much longer than you intended. If your request is serialized, then request pending is like having the other party pick up the telephone and say immediately, "Can you hold, please?"

The Component Busy Condition

A component may reject your request because it has a modal dialog box open, or because a user edit operation is in progress. For example, Microsoft Excel rejects requests from a client application while a spreadsheet cell is being edited.

Visual Basic assumes that the busy condition is temporary and keeps trying the request for a specified timeout interval. When that time is up, Visual Basic displays the Component Busy dialog box, as shown in Figure 10.6.

Figure 10.6 The Component Busy dialog box


Component Busy

An action cannot be completed because a component (Microsoft Excel) is busy. Choose "Switch To" to activate the component and correct the problem.

The user can retry the request, cancel the request, or switch to the component and fix the problem (for example, by dismissing the dialog box). If the user chooses Cancel, the error &h80010001 (RPC_E_CALL_REJECTED) is raised in the procedure that made the request.

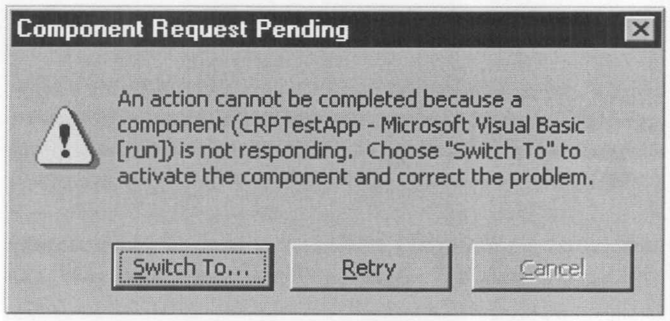
The Request Pending Condition

Once a component has accepted your application's request, your application must wait until the request has been completed. If the request takes a long time, the user may attempt to minimize your program or resize it to get it out of the way.

After a short timeout interval, Visual Basic responds to such attempts by displaying the Component Request Pending dialog.

The appearance of the Component Request Pending dialog box is slightly different from the Component Busy dialog. The Cancel button is disabled, as shown in Figure 10.7, because the request in progress cannot be canceled.

Figure 10.7 The Component Request Pending dialog box


Component Request Pending

An action cannot be completed because a component (CRPTestApp - Microsoft Visual Basic [run]) is not responding. Choose "Switch To" to activate the component and correct the problem.

Switching to the component is useful only if it has halted to display an error message as a result of your request. This should not be a common occurrence, because the proper behavior for a component is to return an error condition to the program that called it.

For More Information For more information, see "Changing the Component Busy or Request Pending Messages," "Controlling Timeout Intervals," and "Raising an Error on Component Busy Timeout" later in this chapter.

Changing the Component Busy or Request Pending Messages

The Component Busy and Component Request Pending dialog boxes are provided by Visual Basic as simple default messages. There are many situations where these dialog boxes may not meet your needs.

- Your program may call a method of an object provided by a component that has no user interface. Components created using Visual Basic, Professional or Enterprise Editions, for example, may run in the background without any visible forms.
- The component you call may have been created using the Remote Automation features of Visual Basic, Enterprise Edition, and may be running on another computer located at some distance from the user.
- If your program has loaded a Microsoft Excel workbook using the GetObject function, the workbook will not be visible when the user switches to Microsoft Excel. In fact, Microsoft Excel itself may not be visible, in which case the Switch To button does nothing.

In these situations, the Switch To button is inappropriate and may confuse the user of your program. You can specify a substitute message for either or both of the timeouts. Your messages will be displayed in a simple message box, without a Switch To button.

For the request pending condition, the message box has only an OK button. For the component busy condition, an OK button and a Cancel button are provided. If the user presses Cancel, error -2147418111 (&h80010001) will be raised in the procedure in which you made the request.

The following properties of the App object determine whether the Component Busy or Component Request Pending dialog box will be replaced by a message box and allow you to specify the text and caption of the message box.

OLEServerBusyMsgText Property

Specifies the message text to be displayed when the component busy condition occurs. Setting this property causes the alternate message box to be used in place of the usual Component Busy dialog box.

OLEServerBusyMsgTitle Property

Specifies the caption to be used if an alternate message is supplied for the component busy condition. (Only setting this property will not cause the alternate message box to be used.)

OLERequestPendingMsgText Property

Specifies the message text to be displayed when the request pending condition occurs. Setting this property causes the alternate message box to be used in place of the usual Component Request Pending dialog box.

OLERequestPendingMsgTitle Property

Specifies the caption to be used if an alternate message is supplied for the request pending condition. (Only setting this property will not cause the alternate message box to be used.)

The following example sets titles and message texts for both the component busy and pending request conditions, completely overriding the Component Busy and Component Request Pending dialog boxes.

```
Public Const APP_TITLE = "Demo Application"
```

```
Private Sub cmdLongTransaction_Click()
   On Error Goto LongTransaction_Error
   ' You may wish to set the titles once, in Sub Main.
   App.OLEServerBusyMsgTitle = APP_TITLE
   App.OLERequestPendingMsgTitle = APP_TITLE
   ' Message texts specific to this request.
   App.OLEServerBusyMsgText = "The component for _
      the " & "Long Transaction has not responded. _
   If " & "you have been waiting more than five " _
        & "minutes, you may wish to cancel this " _
        & "request and try it later." & vbCrLf _
        & "Call Network Services to verify that the " _
        & "component is running, or to report problems."
   App.OLERequestPendingMsgText = "Your request " _
        & "is still executing. " & vbCrLf _
        & "Call Network Services to verify that the " _
        & " component is running, or to report problems."
   ' Code to make a request and use results...
   ' ...
LongTransaction_Cleanup:
   ' Code to perform any necessary cleanup...
   ' ...
   Exit Sub

LongTransaction_Error:
   If Err.Number = &h80010001 Then
      MsgBox "Transaction cancelled"
   Else
      ' Code to handle other errors.
   End If
   Resume LongTransaction_Cleanup
End Sub
```

Important The length of your messages may be limited by the operating system. Messages more than a thousand characters in length can be used when the target operating system is Windows NT or Windows 95.

Controlling Timeout Intervals

You can set the timeout intervals that determine when Visual Basic displays the Component Busy and Component Request Pending dialog boxes, using two properties of the App object.

OLEServerBusyTimeout Property

Determines how long Visual Basic will go on retrying your Automation requests before displaying the Component Busy dialog. The default value is 10000 milliseconds (10 seconds).

OLERequestPendingTimeout Property

Determines how long Visual Basic waits before responding to mouse clicks, keypress events, and other events by displaying the Component Request Pending dialog. The default value is 5000 milliseconds (5 seconds).

The following example shows how the timeout values might be adjusted and reset for a call to the StockAnalysis method of a hypothetical BusinessRules object.

```
Public Sub SetTimeouts(ByVal lngComponentBusy As _
 Long, ByVal lngRequestPending As Long)
   App.OLEServerBusyTimeout = lngComponentBusy
   App.OLERequestPendingTimeout = lngRequestPending
End Sub


Public Sub ResetTimeouts()
   App.OLEServerBusyTimeout = 10000

   App.OLERequestPendingTimeout = 5000
End Sub


Private Sub cmdFullAnalysis_Click()
   On Error Goto FullAnalysis_Error
   ' Set very short timeouts. After 2 seconds,
   ' the user will be notified and keypresses or
   ' clicks will display the Component Busy
   ' and Component Request Pending dialogs.
   SetTimeouts 2, 2
   Me.MousePointer = vbHourglass
   gobjBusinessRules.StockAnalysis txtNYSECode.Text, ATYPE_FULL
FullAnalysis_Cleanup:
   Me.MousePointer = vbDefault
   ResetTimeouts
   Exit Sub


FullAnalysis_Error:
   If Err.Number = &h80010001 Then
      MsgBox "Analysis cancelled"
   Else
      ' Code to handle other errors...
   End If
   Resume FullAnalysis_Cleanup
End Sub
```

You can set either of these timeouts to very large values, because they are stored as Longs. For example, 86,400,000 milliseconds is a day, which is equivalent to an infinite timeout. When you do this, however, you risk having your program lock up until the component is no longer busy, or until a pending request has completed.

Important Because these timeout values are properties of the App object, they also affect documents you link or embed using the OLE container control or the Toolbox. If you are using linked or embedded documents and you change these properties for an Automation request, it is a good idea to reset the values afterward.

Raising an Error on Component Busy Timeout

For the component busy condition, you can bypass both the Component Busy dialog box and the replacement message by setting the Boolean OLEServerBusyRaiseError property of the App object to True. Visual Basic will retry your request for the length of time specified by the OLEServerBusyTimeout property, and then raise an error in the procedure that made the Automation request, just as if the user had pressed the Cancel button on the Component Busy dialog box.

The error returned is -2147418111 (&h80010001). In the error handler for the procedure, you can then take whatever action is most appropriate. For example, you could display a complex dialog box that offered the user several retry options or alternatives.

This property will be particularly useful for components designed to run on remote network computers, using the Remote Automation feature of Visual Basic, Enterprise Edition. Such a component may call on other components, and it must handle errors in those calls without displaying any forms.

There is no corresponding property for the request pending condition. Once an Automation request has been accepted by the component, the client program must wait until the request is complete.

Using a Component's Visual Interface

If a component supports object linking and embedding (OLE), you can link or embed an object into your application without writing any code by using the component's visual interface. You can use a component's visual interface in one of two ways:

- By adding an OLE container control to your application, then inserting an object into the control.
- By adding the object's class to the Toolbox, then adding an object of that class to your application just as you would add a control to a form.

Inserting an Object with the OLE Container Control

The OLE container control gives you the most flexibility in using an object's visual interface. With the OLE container control, you can:

- Create a placeholder in your application for an object. You can create the object that appears within the OLE container control at run time, or you can change an object you have placed in the OLE container control at design time.
- Create a linked object in your application.
- Bind the OLE container control to a database.
- Perform an action if the user moves, sizes, or updates the object in the OLE container control.
- Create objects from data that was copied onto the Clipboard.
- Display objects as icons.

An OLE container control can contain only one object at a time. There are several ways to create a linked or embedded object in the OLE container control — the one you choose depends on whether you are creating the linked or embedded object at design time or run time. Once you have an OLE container control drawn on your form, you can insert an object into the container control by:

- Using the Insert Object or Paste Special dialog box. See "Inserting Objects at Design Time with the OLE Container Control" and "Creating Objects at Run Time with the OLE Container Control."
- Setting the Class, SourceDoc, and SourceItem properties in the Properties window. See "Creating Objects at Run Time with the OLE Container Control."
- Calling the CreateEmbed or CreateLink method. See "Creating Objects at Run Time with the OLE Container Control."

For More Information For more information on using the OLE container control, see "OLE Container Control" and "Containers for Controls" in Chapter 7, "Using Visual Basic's Standard Controls."

Inserting an Object by Adding Its Class to the Toolbox

In the same way that you use the Toolbox to add one of Visual Basic's built-in controls to an application, you can use the Toolbox to add an object. First, add the object's class to the Toolbox, then add the object to a form.

To add an object's class to the Toolbox

1. From the **Project** menu, choose **Components**.
2. In the **Components** dialog box, click the **Insertable Objects** tab.

3. Select the class you want to add to the Toolbox, then click **OK**. Visual Basic adds a button of that class to the toolbox.
4. For example, to add a Microsoft Excel Worksheet button to the Toolbox, select Microsoft Excel Worksheet.

Once you've added the object's class to the Toolbox, you can draw it on a form to create an object of that class. For example, after you add a Microsoft Excel Worksheet button to the Toolbox, you can draw it on a form to create a worksheet object on the form.

Contrasting Linked and Embedded Objects

You use a component's visual interface to contain data from another application by linking or embedding that data into your Visual Basic application. The primary difference between a linked and embedded object is where their data is stored. For example, data associated with a *linked object* is managed by the application that created it and stored outside an OLE container control. Data associated with an *embedded object* is contained in an OLE container control and can be saved with your Visual Basic application.

When a linked or embedded object is created, it contains the *name* of the application that supplied the object, its data (or, in the case of a linked object, a *reference* to the data), and an *image* of the data.

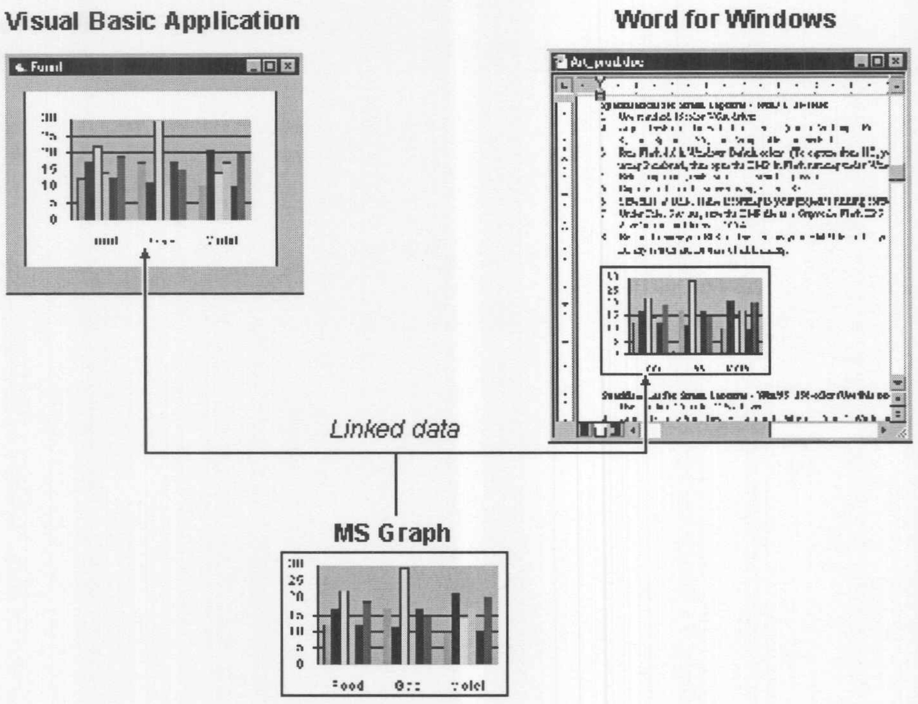
Note To place an object in an OLE container control, the component that provides the object must be registered in your system registry. When you install an application that supplies the objects you want to use in your project, that application should register its object library on your system so that application's objects appear in the Insert Object dialog box. You can use Regedit.exe to search the system registry for an object, but take care not to alter the contents of the registry.

Linked Objects

When you link an object, you are inserting a *placeholder* (not the actual data itself) for the *linked object* into your application. For example, when you link a range of spreadsheet cells to a Visual Basic application, the data associated with the cells is stored in another file; only a link to the data and an image of the data are stored in the OLE container control. While working with your Visual Basic application, a user can activate the linked object (by double-clicking the object, for example), and the spreadsheet application will start automatically. The user can then edit those spreadsheet cells using the spreadsheet application. When editing a linked object, the editing is done in a separate window outside the OLE container control.

When an object is linked to a Visual Basic application, the object's current data can be viewed from any other applications that contain links to that data. The data exists in only one place the ActiveX component which is the source application that provides the object. For example, in Figure 10.8, Visual Basic contains a link to the Graph application. Microsoft Word also contains a link to the graph. If the graph's data is changed by either application, the modified graph will appear in *both* the Visual Basic application and the Microsoft Word document.

Figure 10.8 An object's data can be accessed from many different applications that contain links to that data



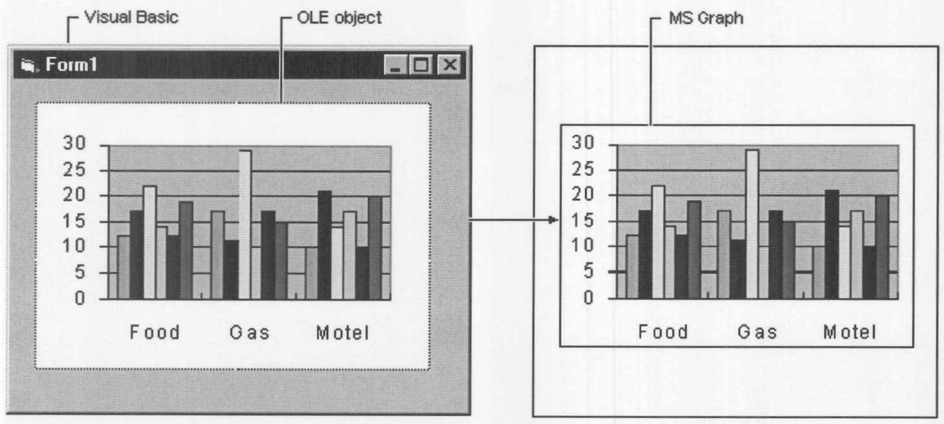
As you can see, linking makes it easy to track identical information that appears in more than one application. Linking is useful when you want to maintain one set of data that is accessed from several applications.

Embedded Objects

To create an embedded object, you can either use an OLE container control or add an object's class to the Toolbox. With an *embedded object*, all the data associated with the object is copied to and contained in the OLE container control. When you save the contents of the control to a file, the file contains the name of the application that produced the object, the object's data, and a metafile image of the object. For this reason, embedded objects can greatly increase file size.

Unlike linked objects, no other application has access to the data in an embedded object. Embedding is useful when you want your application to maintain data that is produced and edited in another application, as shown in Figure 10.9.

Figure 10.9 Your application maintains data for an embedded object

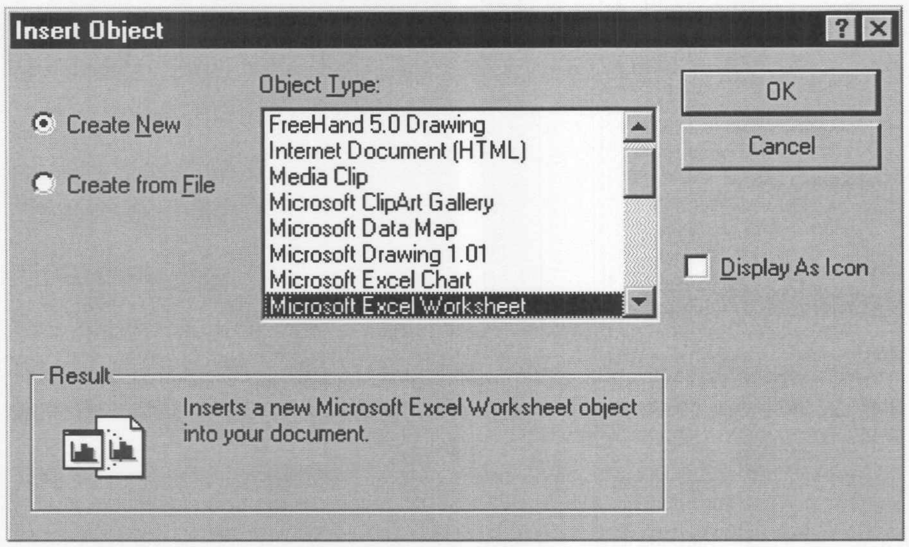


When the user activates the object (the graph), the ActiveX component that created the object (Microsoft Graph) is invoked by the container application (your Visual Basic application), and the object's data is opened for editing. In addition, the user interface and menu system of the object is displayed in the container application so the user can control the object in place. For more information on in-place activation, see "Activating an Object in the OLE Container Control" later in this chapter.

Inserting Objects at Design Time with the OLE Container Control

Each time you draw an OLE container control on a form, Visual Basic displays the Insert Object dialog box. You use this dialog box, shown in Figure 10.10, to insert linked or embedded objects at design time. The Insert Object dialog box presents a list of the available objects you can link or embed into your application.

Figure 10.10 The Insert Object dialog box



Inserts a new Microsoft Excel Worksheet object into your document.

When you insert an object into the OLE container control at design time, the Class, SourceDoc, and SourceItem properties are automatically set for you. These properties identify the application that supplies the object, the source file name, and any specific data that is linked from within that file. For more information about these and other properties and methods that apply to the OLE container control, see "Inserting Objects at Run Time."

Inserting Linked Objects at Design Time

When you insert a linked object, the data displayed in the OLE container control exists in one place — the source file. The object's current data can be viewed from any other applications that contain links to that data. The OLE container control maintains the object's link information, such as the name of the application that supplied the object, the name of the linked file, and an image of the linked data.

To insert a linked object using the Insert Object dialog box

1. Draw an **OLE container control** on a form.
2. The **Insert Object** dialog box is displayed. You can also display this dialog box at any time by clicking the **OLE container control** with the right mouse button and then choosing the **Insert Object** command.
3. Select the **Create from File** option button.
4. Choose the **Browse** button.
5. A **Browse** dialog box is displayed.
6. Select the file you want to link.
7. Click **Insert** to return to the **Insert Object** dialog box.
8. Select the **Link** check box in the **Insert Object** dialog box, and choose **OK** to create the linked object.

When you use a linked object, every user who runs your application must have access (a valid path) to the linked file and a copy of the application that created the file. Otherwise, when your application is run, an image of the original data is displayed, but the user will not be able to modify the data, nor will the user see changes others may have made to the linked data. This may be a concern if your application is running on a network.

If your application contains a linked object, it is possible for that object's data to be changed by another application when your application is not running. The next time your application is run, changes to the source file do not automatically appear in the OLE container control. To display the current data in the OLE container control, use the control's Update method:

```
oleObj.Update
```

For More Information See "Update Method (OLE Container)" in the *Language Reference* in Books Online.

If a user wants to save changes to a linked object, the user must save it from the ActiveX component's menu. The OLE container control's SaveToFile method applies only to embedded objects.

Creating Embedded Objects at Design Time

When you create an embedded object, you can either embed data from a file or create a new, empty object that can be filled with data later. When you embed data from a file, a copy of the specified file's data is displayed in the OLE container control. When you create a new object, the application that created the object is invoked and you can enter data into the object.

Typically, you create embedded objects that display existing data at design time. This allows you to view the object's data as it will appear to the user. You can then move and size the OLE container control and the other controls on the form to create your application's user interface.

To display existing data in an embedded object, create the object using an existing file as a template. The OLE container control then contains an image of the data in the file. An application that displays data using an embedded object will be larger than an application that displays the same data using a linked object, because the application with the embedded object actually contains the source file's data.

To create an embedded object using an existing file

1. Draw an **OLE container control** on your form.

1. The **Insert Object** dialog box is automatically displayed.

1. Select the **Create from File** option button.
2. Choose the **Browse** button.

1. A **Browse** dialog box is displayed.

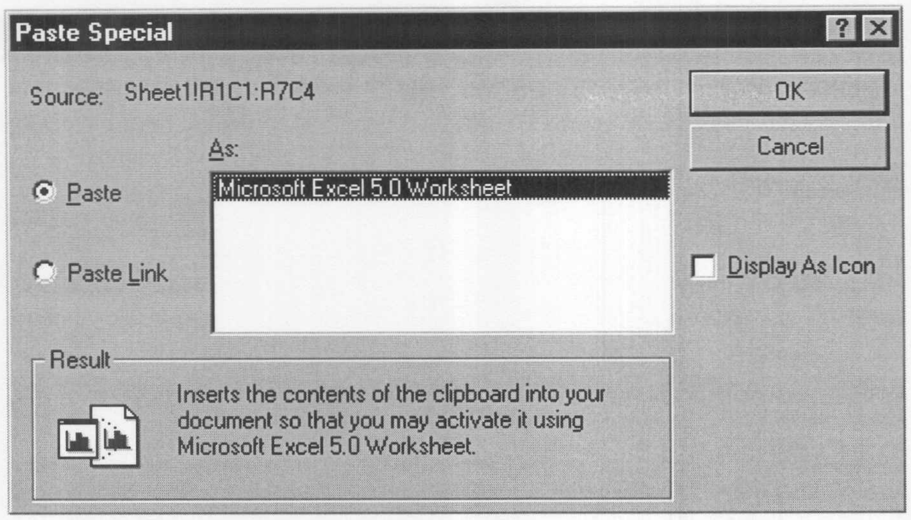
1. Select the file you want to embed.
2. Choose Insert to return to the **Insert Object** dialog box.
3. In the **Insert Object** dialog box, choose **OK** to create the embedded object.

Unlike the data in a linked object, data in an embedded object is not persistent. In other words, if you want changes entered by the user to appear the next time your application is run, you must use the SaveToFile method to save the data. For more information on saving embedded data to a file, see "Saving and Retrieving Embedded Data" later in this chapter.

Creating Objects Using the Paste Special Dialog Box

Another way to create an object at design time is to use the Paste Special dialog box (shown in Figure 10.11). This dialog box is helpful if you only want to use a portion of a file — for instance, a range of cells from a spreadsheet, or a paragraph from a Word document.

Figure 10.11 The Paste Special dialog box



Inserts the contents of the clipboard into your document so that you may activate it using Microsoft Excel 5.0 Worksheet.

To create an object using the Paste Special dialog box

1. Run the application containing the data you want to link or embed.
2. Select the data you want to link or embed.
3. From the ActiveX component's **Edit** menu, choose **Copy**.
4. The data is copied onto the Clipboard.
5. In Visual Basic, click the **OLE container control** with the right mouse button, and choose the **Paste Special** command from the pop-up menu.
6. Select the **Paste** option button if you want to create an embedded object.
   or
   Select the **Paste Link** option button if you want to create a linked object.
7. If there is already an object embedded or linked in the control, a message asks whether you'd like to delete that existing object and create a new one in its place.
8. Choose **OK** to create the object.

Creating Objects at Run Time with the OLE Container Control

To create a linked or embedded object at run time, you use methods and properties in code. The OLE container control has a variety of properties and methods that you can use for manipulating linked or embedded objects. For a complete list of the properties and methods that apply to the OLE container control, see "OLE Container Control" in the *Language Reference* in Books Online.

Using the Object Property

By using the OLE container control's Object property, you can also use the properties and methods of the linked or embedded object. The Object property is a run-time, read-only property that holds a reference to the object in an OLE container control. Use this property to perform Automation tasks with the OLE container control, including programmatically manipulating the properties and methods an object supports:

```
strObjName = oleObj1.Object.Name
```

To use this property, the OLE container control must contain an object that is programmable. For more information on programmable objects, see "Types of ActiveX Components."

Creating Linked Objects at Run Time

You can create a linked object from a file at run time with the OLE container control's CreateLink method. This method takes one argument, *sourcedoc*, which is the file from which the object is created, and an optional argument, *sourceitem*, which specifies the data you want to link from within the source file. The following code fragment creates a linked object at run time:

```
oleObj1.CreateLink "C:\Excel\Test.xls"
```

Note If you use CreateLink to create a linked object, you do not have to set the Class, SourceDoc, and SourceItem properties in the Properties window.

For More Information See "CreateLink Method" in the Language Reference in Books Online.

Creating Embedded Objects at Run Time

To create an embedded object from a file at run time, you can use the CreateEmbed method. This method has two arguments, *sourcedoc* and *class* (which is optional if SourceDoc is specified). *Sourcedoc* determines the template for the object, and *class* determines the object type. When you use CreateEmbed, you do not need to set the SourceDoc and Class properties.

The following code fragment creates an embedded object using an existing file as a template for the object.

```
oleObj1.CreateEmbed "Q1profit.xls"
```

For More Information See "CreateEmbed Method" in the Language Reference in Books Online.

When you create an empty embedded object, it is a good idea to activate the ActiveX component that will provide data for the object. You can do this with the DoVerb method. This allows the user to enter any data into the application at run time. The user can then show this newly entered data in the OLE container control by choosing the ActiveX component's Update command (this menu command should appear on the component's File menu).

To create an empty embedded object at run time

1. Use the CreateEmbed method without specifying a source document to create an empty embedded object. For example, this code fragment inserts a file template for a Microsoft Excel Worksheet in the OLE container control:

   ```
   oleObj1.CreateEmbed "","Excel.Sheet"
   ```

2. Use the DoVerb method. The default verb for the DoVerb method depends on the application. With Microsoft Excel, the default verb is Edit.

For example, the following code fragment creates an empty embedded object and then activates the application that created it using the default DoVerb action.

```
oleObj1.CreateEmbed "", "Excel.Sheet"
oleObj1.DoVerb -5 ' Activate
```

Providing empty embedded objects is useful when creating a document-centered application that uses a variety of information from different applications. For more information, see "Letting the User Specify Objects at Run Time."

Binding a Database to the OLE Container Control

You can bind the OLE container control to data stored in the Microsoft Jet database engine or Microsoft Access database. You may want to do this, for example, if you have a database with a table of employee pictures. If the pictures are stored as objects, you can bind them to the OLE container control and display them on a form as each record is accessed with the data control. To bind data to one of these databases, specify the source of data (recordset name) in the DataSource property and the field name from that data source in the DataField property of the OLE container control. When displaying an object from a database, the OLE container control allows the user to activate, edit, and update the object. As with any bound control, the updated object is automatically written back to the database when the record position is changed.

For More Information See Chapter 14, "Accessing Data," or see the *Guide to Data Access Objects*, included with the Professional and Enterprise editions.

Letting the User Specify Objects at Run Time

By displaying the Paste Special and Insert Object dialog boxes at run time, you can allow the user to create a variety of objects. You may do this when creating a document-centered application. In such an application, the user combines data from different applications to create a single document. For instance, this application might be a word processor in which the user might enter some text and then embed a spreadsheet and a chart using the Insert Object or Paste Special dialog box.

You use the OLE container control's InsertObjDlg method to display the Insert Object dialog box, or you can use the PasteSpecialDlg method to display the Paste Special dialog. These two dialogs let the user make decisions about what goes into the OLE container control.

- The Insert Object dialog box presents a list of available objects and creates an object based on the user's selection.
- The Paste Special dialog box allows the user to paste an object from the system Clipboard into an OLE container control.

You can display these dialog boxes at run time by calling the appropriate method on an event for instance, a menu's Click event:

```
Private Sub cmdInsert_Click ()
   ' Display Insert Object dialog box.
   oleObj1.InsertObjDlg
   ' Check to make sure object was created with the
   ' OLEType property.
   If oleObj1.OLEType = vbOLENone Then
      MsgBox "Object Not Created."
   End If
End Sub


Private Sub oleObj1_Click ()
   ' Determine if the data contained on the Clipboard
   ' can be pasted into the OLE container control.
   If oleObj1.PasteOK Then
      ' Display Paste Special dialog box.
      oleObj1.PasteSpecialDlg
      ' Check to make sure object was created.
      If oleObj1.OLEType = vbOLENone Then
         MsgBox "Object Not Created."
      End If
   End If
End Sub
```

Once the dialog box is displayed, you do not need to write more code to create the object. The user makes choices in the dialog box and chooses OK to create an object. If the user cancels a dialog, an object is not created.

Note Before displaying the Insert Object or Paste Special dialog box, you may want to determine the value of the OLEType property to see if the OLE container control contains a linked object, embedded object, or no object, as demonstrated in the preceding code example.

The constant vbOLENone and other intrinsic constants are listed in the Visual Basic (VB) object library of the Object Browser.

Determining How an Object is Displayed in the OLE Container Control

You can use the OLE container control's DisplayType property to indicate if the object will appear as an icon (set DisplayType = 1), or if the object's data will be displayed in the control (set DisplayType = 0). This property also determines the default setting of the Display As Icon check box when the Insert Object and Paste Special dialog boxes are displayed at both run time and design time.

Note Once the OLE container control contains an object, you cannot change its display type. You can, however, delete the linked or embedded object, set the DisplayType property, and then insert a new object.

You use the SizeMode property to determine how an object's icon or data image is displayed in the OLE container control when the control is not UI (user-interface) active. A setting of 0-Clip or 3-Zoom clips the image to fit the control, but it doesn't change the actual size of the image (you might not see all of the image when editing it). An object that is smaller than the control is edited in an area smaller than the control. An object larger than the control fills the entire container area and may be clipped if it is larger than the control area. Alternately, setting SizeMode to 2-AutoSize resizes the control to fit the image.

Activating an Object in the OLE Container Control

While the OLE container control's DoVerb method activates an object at run time, you can use the AppIsRunning property to determine whether the application supplying the object is activated and running. You can set AppIsRunning to True to start the ActiveX component, which causes objects to activate more quickly. You can also set this property to False to close the application or take another appropriate action when the object loses focus.

In-Place Activation

Some embedded objects can be edited (activated) from within the OLE container control. This is called *in-place activation*, because users can double-click an object in your application and interact with application supplying the object, without switching to a different application or window.

For objects that support in-place activation, you can set the AutoActivate property so that users can activate an object at any time. That is, when the OLE container control's AutoActivate property is set to Double-Click, users can double-click the control to activate it. It is important to note that activating an object launches that object's application if it is not already running.

Note If you want to display the ActiveX component's menus at run time when the user clicks the OLE container control, you must define at least one menu item for the form and set its Visible property to False. This can be an invisible menu if you don't want any menus displayed. See Chapter 6, "Creating a User Interface," for more information on displaying an ActiveX component's menus and toolbars in a container application when an object is activated at run time.

Responding to Moving or Sizing the Container

The OLE container control has the ObjectMove event, which is triggered when the object associated with the OLE container control is moved or resized. The arguments to ObjectMove represent the coordinates of the object (excluding its border) within the object's container. If the object is moved off the form, the arguments have values representing the position relative to the upper-left corner of the form. These can be positive or negative. If the Width or Height of the ActiveX component is changed, the OLE container control is notified.

The ObjectMove event is the only way the OLE container control can determine if the object has been moved or resized. An ObjectMove event occurs when the user moves or resizes the object contained in the OLE container control. For example:

```
Private Sub oleObj1_ObjectMove(Left As Single, Top As _
   Single, Width As Single, Height As Single)
   ' This method resizes the OLE container control to
   ' the new object size.
   oleObj1.Move oleObj1.Left, oleObj1.Top, Width, Height
   ' This method moves the OLE container control
   ' to the new object position.
   oleObj1.Move Left, Top, oleObj1.Width, oleObj1.Height
' Repaints the form.
   Me.Refresh
End Sub
```

Saving and Retrieving Embedded Data

Data associated with an embedded object is not persistent; that is, when a form containing an OLE container control is closed, any changes to the data associated with that control are lost. To save updated data from an object to a file, you use the OLE container control's SaveToFile method. Once the data has been saved to a file, you can open the file and restore the object.

If the object is linked (OLEType = 0-Linked), then only the link information and an image of the data is saved to the specified file. The object's data is maintained by the application that created the object. If a user wants to save changes to a linked file, the user must choose the Save command from the ActiveX component's File menu because the SaveToFile method applies only to embedded objects.

If the object is embedded (OLEType = 1-Embedded), the object's data is maintained by the OLE container control and can be saved by your Visual Basic application.

Objects in the OLE container control can be saved only to open, binary files.

To save the data from an object to a file

1. Open a file in binary mode.
2. Use the SaveToFile method.

The cmdSaveObject_Click event procedure illustrates these steps:

```
Private Sub cmdSaveObject_Click ()
   Dim FileNum as Integer
   ' Get file number.
   FileNum = FreeFile
   ' Open file to be saved.
   Open "TEST.OLE" For Binary As #FileNum
   ' Save the file.
   oleObj1.SaveToFile FileNum
   ' Close the file.
   Close #FileNum
End Sub
```

Once an object has been saved to a file, it can be opened and displayed in an OLE container control.

Note When you use the SaveToFile or ReadFromFile methods, the file position is located immediately following the object. Therefore, if you save multiple objects to a file, you should read them in the same order you write them.

To read data from a file into an OLE container control

1. Open the file in binary mode.
2. Use the ReadFromFile method on the object.

The cmdOpenObject_Click event procedure illustrates these steps:

```
Private Sub cmdOpenObject_Click ()
   Dim FileNum as Integer
   ' Get file number.
   FileNum = FreeFile
   ' Open the file.
   Open "TEST.OLE" For Binary As #FileNum
   ' Read the file.
   oleObj1.ReadFromFile FileNum
   ' Close the binary file.
   Close #FileNum
End Sub
```

The Updated event is invoked each time the contents of an object is changed. This event is useful for determining if an object's data has been changed because it was last saved. To do this, set a global variable in the Updated event indicating the object needs to be saved. When you save the object, reset the variable.